

Problem Set 4—Due Friday, March 2. 11AM

This problem set is intended to get you started on approximate solutions to NP-hard problems. There will be a little followup work on this in the take home final (which will build on your solutions to this one).

The problem we will be addressing is **vertex cover**: given an undirected graph $G = (V, E)$ we want to find a minimum size set of vertices which cover all edges in E (see section 34.5.2 and 35.1 of the text). Since vertex cover is NP-hard we can't expect to solve it exactly and quickly on all inputs, but we can hope to get close to solving it. In this assignment we will try some variants of the approximation algorithm described in section 35.1 of the text.

A):**1.** To start with, use the code from Ps2 that builds a random undirected graph. This takes in two parameters: n the number of vertices in the graph, and p the percentage of expected edges in the graph. Thus if $n = 200$ and $p = 10$ you want to build a 200 vertex random graph where each edge (i, j) exists with probability $10/100 = .1$.

You can use a simple adjacency matrix representation of the graph or one of the adjacency list representations (maybe better) if you like (use an n by n character array or a bit array for the adjacency matrix) but don't try values of $n > 5000$. Let $d(v)$ be the degree of vertex v , which is the number of edges which touch it.

2. Implement the simple vertex cover algorithm described in 35.1 on the random graphs you generate using the following methods.

1) pick arbitrary edges in step 4.

Try a possibly better heuristic:

2) Choose the edge (u, v) such that $d(u) + d(v)$ is as large as possible (where the degrees are computed in the current graph, that is the one where the covered edges are deleted).

3. Try to make your solutions to part 2 as fast as possible. (you may change the graph representation if you like)

4. If you aren't already doing this, after finding a cover do a pass which removes all redundant vertices (e.g. if vertex 2 is in the cover as well as all neighbors of 2, you can delete 2 and still have a cover).

Also, add code which checks that the cover you find is valid: go through the edges and make sure at least one end point is in the cover.

5. Implement a simulated annealing, other neighborhood search, or branch and bound algorithm for vertex cover (you can adjust simulated annealing to run on most size problems by changing the iteration count and/or cooling schedule. for B&B you may not be able to do the whole tree for larger problems, but you can still report the best solution found within some time limit, and how far it can be from optimal). Extra credit: implement more than one method.

6. Try the approximation algorithms above on at least 10 graphs. Make n as large as possible (such that you can solve problems in under a minute, but keep n below 5000), and then solve at least two graphs each for values of $p = 1, 10, 30, 50,$ and 70 .

What to turn in: (A) The code for all your algorithms.

(B) A high level description of your solutions (what data structures you used, what were the bottlenecks you addressed).

(C) An analysis of the run time of your solutions to parts 2/3.

(D) A summary of your experiments: size of solution for each value of p (and each approach) and how long it took to run.

(E) Your analysis of the approximation algorithm results (how well are your algorithms doing, and which one(s) would you choose)?

Paper and Pencil Questions

(F) Dynamic programming can help to reduce the (exponential) time to solve hard problems. For example, for the Traveling Salesman problem a simple brute force approach tries all tours and picks the shortest. There are $(n - 1)!$ tours in an n vertex graph, which makes this approach too slow even for $n = 20$. Using dynamic programming we can solve the problem in $O(n^3 2^n)$ time as follows.

The basic values we compute are for triples (A, s, d) where A is a subset of the vertices, and s, d are two distinct vertices in A . For each possible A, s, d triple we compute $P(A, s, d)$, the shortest path from s to d which visits every vertex in A exactly once (and no other vertices).

To compute the $P(A, s, d)$ values we start with all sets A of size two (so there are only two s, d ordered pairs, and only one (length one) path for each such pair). Show the following:

1) For a general step, assume you know $P(A, s, d)$ for all sets A of size k . Show how to compute the $P(A, s, d)$ for all sets A of size $k + 1$.

2) Once you have computed all the $P(A, s, d)$ values for sets of size n , how do you compute the length of an optimal tour (a cycle which visits each vertex once)?

3) analyze the run time of your algorithm to show it achieves the $O(n^3 2^n)$ time.

(G) The *Steiner Tree problem* is defined as follows: you are given an undirected weighted graph $G = (V, E)$ with $V = R \cup S$. You want to find a minimum cost tree which connects all vertices in R , but you may also use vertices in S as helpful intermediate vertices (but you are not required to use any vertex in S). For example if u, v were vertices in R , x in S and we had arcs $(u, v), (u, x), (x, v)$ of cost 20,10,6; then we would never use the arc (u, v) , always preferring to use the arcs $(u, x), (x, v)$.

The Steiner tree problem is NP-hard. A simple approximation algorithm is to just find a Minimum Spanning Tree (MST) on R . Give an example which shows that this may give an arbitrarily bad solution (i.e. for any constant k you can find a setting where the MST cost is more than k times the optimal cost).

As an alternative, we might like to use a branch-and-bound type of approach to the problem. Decisions would be of the form: use vertex x in S or don't use vertex x (since vertices in R are required we wouldn't make decisions on them). Describe a method to get lower bounds on nodes in the Branch-and-bound tree after having made a sequence of use and don't use decisions (hint: each used vertex must have at least one adjacent edge). Your goal is to get a LB which is fairly tight (e.g. a lower bound of zero is correct but not useful).